CSE113: Parallel Programming

Jan. 31, 2022

- **Topics**:
  - Intro to concurrent data structures
  - Bank account example
  - Specification: Sequential consistency

Announcements

- Expect HW1 grades by Monday
  - Afterwards, please let us know about any issues ASAP

- Homework 2 is due on Tomorrow
  - You have 4 free late days

- Homework 3 is released on Monday
  - 10 days + 4 free late days

- Midterm is released on Monday
  - asynchronous, Monday morning to Friday night
  - no time limit
  - Open note, open internet (to a reasonable extent: no googling exact questions or asking questions on forums or chatGPT)
  - do not discuss with classmates AT ALL while the test is active
  - **No late tests will be accepted.**

Previous quiz

Which of the following are NOT ways that mutex implementations can encourage fair access?

○ Sleeping

○ Yielding

○ Using a ticket lock

○ relaxed peeking

Previous quiz

A reader-writer mutex allows multiple readers in the critical section, mutliple writers in the critical section, but never a combination of readers and writers.

○ True

○ False

Previous quiz

If you are an expert in how your code will compile to machine instructions, it is okay to have data conflicts in your code.

○ True

○ False

Previous quiz

With this being the end of module two, please write a few sentence about how you found this module. For example, was the material clear? was the material interesting? What did you find surprising? What was something that was unclear?

Final thoughts on Module 2

Data conflicts

- Data conflicts are undefined
  - Compiler can do crazy things
  - interleavings cause bugs that are extremely rare

- Your code should use mutexes to avoid data conflicts!

- What happens when you don't?

Horrible data conflicts in the real world

Therac 25: a radiation therapy machine

- Between 1987 and 1989 a software bug caused 6 cases where radiation was massively overdosed

- Patients were seriously injured and even died.

- Bug was root caused to be a data conflict.

- https://en.wikipedia.org/wiki/Therac-25

- https://www.youtube.com/watch?v=Ap0orGCiou8

Horrible data conflicts in the real world

2003 NE power blackout

- second largest power outage in USA history: 55 million people were effected

- NYC was without power for 2 days, estimated 100 deaths

- Root cause was a data conflict

- https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

But checking for data conflicts is hard…

- Tools are here to help (Professor Flanagan is famous in this area)

How do they work?

- Two approaches

- **Happens-before**: build a partial order of mutex lock/unlocks. Any memory access that can't be ordered in this partial order is a conflict.

- **Lockset**: Every shared memory location has is associated with a set of locks. Refine the lockset for every access and evaluate the final result.

Dynamic Analysis

- Thread sanitizer:
  - a compiler pass built into Clang
  - About 10x overhead when you run the program
  - Identifies data conflicts and deadlocks

Static Analysis

- Facebook Infer:
  - Statically checks for many issues (memory safety, assertions)
  - Can check for races in concurrent classes
  - Main support is for Java, although they claim support for C++

Current state of data conflicts

- A recent tool:
  - Checks for C++ races
  - Scales to large programs

- Reports:
  - Chrome has 6 unresolved data-conflicts
  - Firefox has 52 unresolved data-conflicts

- Difficult to fix! 6.7 million lines of code in Chrome
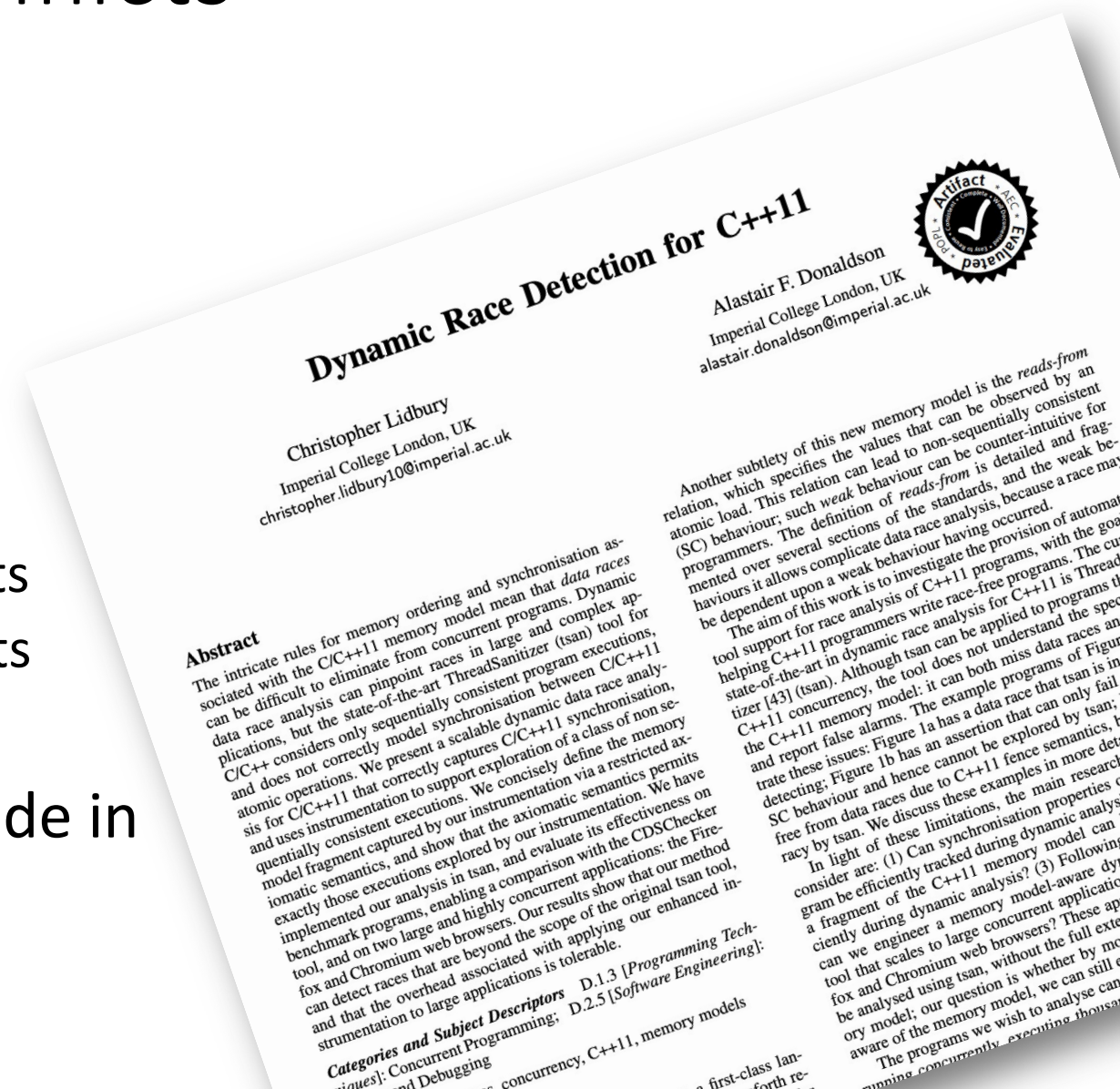
Summary

- Avoid data conflicts! They can cause serious bugs that trigger very very very rarely. (heisenbugs).
  - Better to use too many mutexes than not enough

- Use tools to help you!
  - Infer can helps with Java
  - Thread sanitizer helps with C++

On to new stuff!

Schedule

- Intro to concurrent data structures

- Bank account example

- Specification: Sequential consistency

Concurrent object motivation

- Programming basics cover a set of primitives:
  - types: ints, floats, bools
  - functions: call stacks, recursion

Concurrent object motivation

- Programming basics cover a set of primitives:
  - types: ints, floats, bools
  - functions: call stacks, recursion

simple example:
We can understand this!

```c
//Fibonacci Series using Recursion
#include<stdio.h>
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}
```

https://www.geeksforgeeks.org/c-program-for-n-th-fibonacci-number/

Concurrent object motivation

- How does it look moving into a more complicated setting?

Concurrent object motivation

- How does it look moving into a more complicated setting?
  - Hello world Android app:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d("MainActivity", "Hello World");
}
```

https://developer.android.com/codelabs/android-training-hello-world#7

Concurrent object motivation

- How does it look moving into a more complicated setting?
  - Hello world Android app:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d("MainActivity", "Hello World");
}
```

what the heck is a bundle?

https://developer.android.com/codelabs/android-training-hello-world#7

Concurrent object motivation

- How does it look moving into a more complicated setting?
    - Hello world Android app:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d("MainActivity", "Hello World");
}
```

what is this?

https://developer.android.com/codelabs/android-training-hello-world#7

Concurrent object motivation

- How does it look moving into a more complicated setting?
  - Hello world Android app:

- These are objects!

https://developer.android.com/codelabs/android-training-hello-world#7

Concurrent object motivation

- Objects are user-specified abstractions:
  - A collection of data (state) and methods (behavior) representing something more complicated than primitive types can express.

Concurrent object motivation

- Objects are user-specified abstractions:
  - A collection of data (state) and methods (behavior) representing something more complicated than primitive types can express.

- Examples:
  - Writing a video game? objects for enemies and players
  - Writing an IOS app? objects for buttons

Concurrent object motivation

- Objects are user-specified abstractions:
  - A collection of data (state) and methods (behavior) representing something more complicated than primitive types can express.

- Examples:
  - Writing a video game? objects for enemies and players
  - Writing an IOS app? objects for buttons

- Objects allow programmer productivity:
  - Modular
  - Encapsulation
  - Composable

Concurrent object motivation

- Objects are user-specified abstractions:
  - A collection of data (state) and methods (behavior) representing something more complicated than primitive types can express.

- Examples:
  - Writing a video game? objects for enemies and players
  - Writing an IOS app? objects for buttons

- Objects allow programmer productivity:
  - Modular
  - Encapsulation
  - Composable

- We would like objects in the concurrent setting!

Concurrent object motivation

- Note:
  - The foundations in this lecture are general, and can be widely applied to many different types of objects

  - We will focus on "container" objects, lists, sets, queues, stacks.

  - These are:
    - Practical - used in many applications
    - Well-specified - their sequential behavior is agreed on
    - Interesting implementations - great for us to study!

Conceptual examples

- Shopping list: Going shopping with roommates

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

Consider two people splitting the work.

Conceptual examples

- Shopping list: Going shopping with roommates

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

What can go wrong?

Consider two people splitting the work.

Conceptual examples

- Shopping list: Going shopping with roommates

Best case:
2x as fast (so we can get back to CSE113 homework)

What can go wrong?

We end up with duplicates

eggs
carrots
tortillas

Consider two people splitting the work.

Conceptual examples

- Shopping list: Going shopping with roommates

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

What can go wrong?

We end up with duplicates

We end up missing an item

Consider two people splitting the work.

Conceptual examples

- Shopping list: Going shopping with roommates

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

What can go wrong?

We end up with duplicates

We end up missing an item

If my roommate decides to go surfing, then I could get stranded!

Consider two people splitting the work.

Conceptual examples

- Shopping list: Going shopping with roommates

What kind of object is the list?

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

What can go wrong?

We end up with duplicates

We end up missing an item

If my roommate decides to go surfing, then I could get stranded!
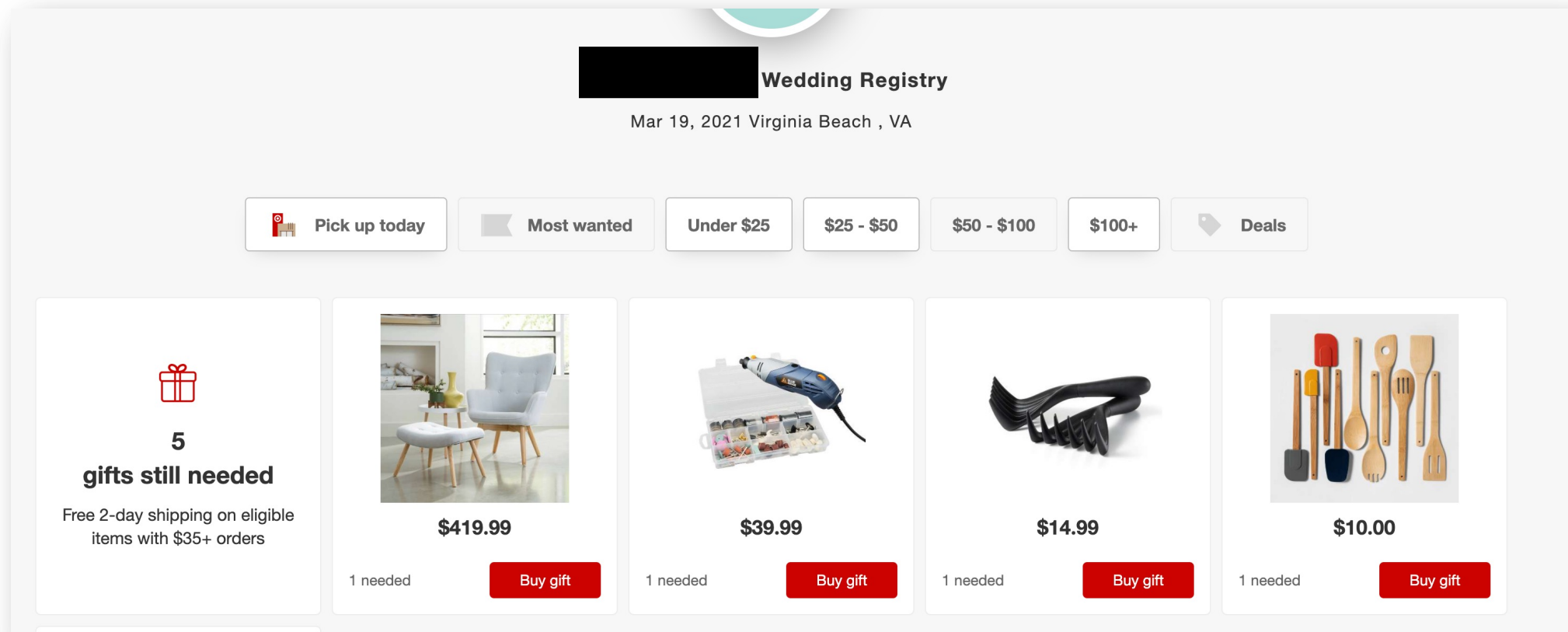
Consider two people splitting the work.

Conceptual examples

- Physically shopping with roommates is a nice conceptual example, but the example also occurs in automated systems

Conceptual examples

- Physically shopping with roommates is a nice conceptual example, but the example also occurs in automated systems

Shared memory concurrent objects

- Lets ground this even more in a shared memory system.

- Shopping cart examples mostly occur in a distributed system setting where there are many different concerns
  - Consider taking a class from Prof. Kuper or Prof. Alvaro!

Shared memory concurrent objects

```
printf("hello world\n");
```

how do we envision printf to work?

```
printf("h");
printf("e");
printf("l");
printf("l");
printf("o");
```
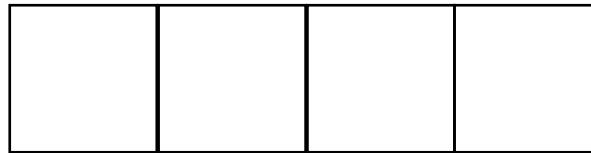
```
terminal:
$ ./a.out




```

Shared memory concurrent objects

```
printf("hello world\n");
```

How does it actually work?

printf("h");
printf("e");
printf("l");
printf("l");
printf("o");

concurrent queue

terminal:
$ ./a.out

./a.out

terminal display

Shared memory concurrent objects

```
printf("hello world\n");
```

How does it actually work?

```
printf("h");
printf("e");
printf("l");
printf("l");
printf("o");
```

concurrent queue

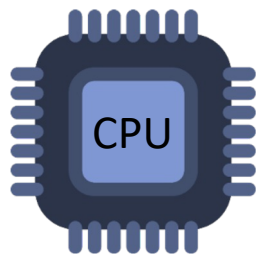|			

./a.out

terminal display

```
terminal:
$ ./a.out
```

You can force a flush with: `fflush(stdout)`

Shared memory concurrent objects

```
printf("hello world\n");
```

Show example

How does it actually work?

```
printf("h");
printf("e");
printf("l");
printf("l");
printf("o");
```

concurrent queue

terminal:
$ ./a.out

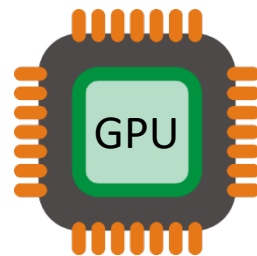./a.out

terminal display

You can force a flush with: `fflush(stdout)`

Shared memory concurrent objects

- Graphics programming

CPU

PCIE

GPU

loop:
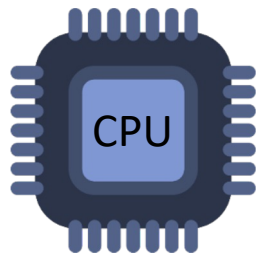   update data (data transfer)
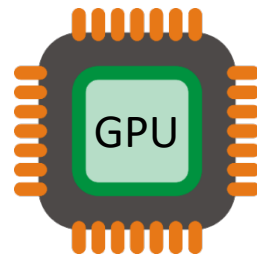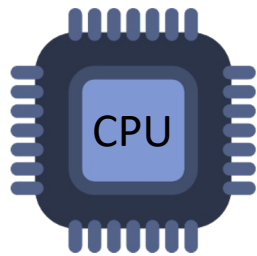   graphics computation (kernel)

Nintendo: breath of the Wild

Shared memory concurrent objects

- Graphics programming

Vulkan/OpenCL CommandQueue

	kernel command	data transfer

PCIE

loop:
    update data (data transfer)
    graphics computation (kernel)

Nintendo: breath of the Wild

Shared memory concurrent objects

- Graphics programming

Vulkan/OpenCL CommandQueue

	kernel command	data transfer

GPU driver concurrently reads from the queue

CPU

PCIE →

GPU

→

loop:
  update data (data transfer)
  graphics computation (kernel)

Nintendo: breath of the Wild

Shared memory concurrent objects

- Graphics programming

Vulkan/OpenCL CommandQueue

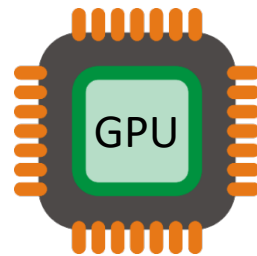|	kernel command	data transfer

GPU driver concurrently reads from the queue

this concurrent queue enables an efficient graphics pipeline
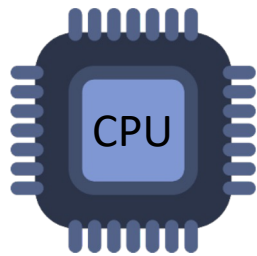
CPU

PCIE

Transferring data for scene 2

GPU

Computation for scene 1

Scene 0

loop:
    update data (data transfer)
    graphics computation (kernel)

Nintendo: breath of the Wild

Shared memory concurrent objects

- Graphics programming

Single writer, single reader
Like in `printf`
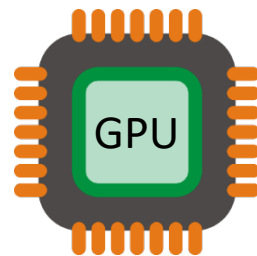
Vulkan/OpenCL CommandQueue

	kernel command	data transfer

GPU driver concurrently reads from the queue

CPU

PCIE

Transferring data for scene 2

GPU

Computation for scene 1

Scene 0

loop:
    update data (data transfer)
    graphics computation (kernel)

Nintendo: breath of the Wild
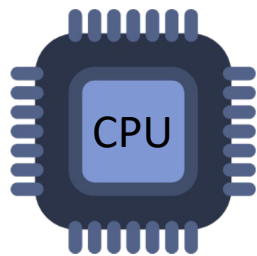
Shared memory concurrent objects

- Graphics programming

Multiple producers

Process 0

GPU Driver

Processes write to the
queue concurrently

| kernel command | kernel command | data transfer | data transfer |

PCIE

GPU

Process 1

Computation
for scene 1
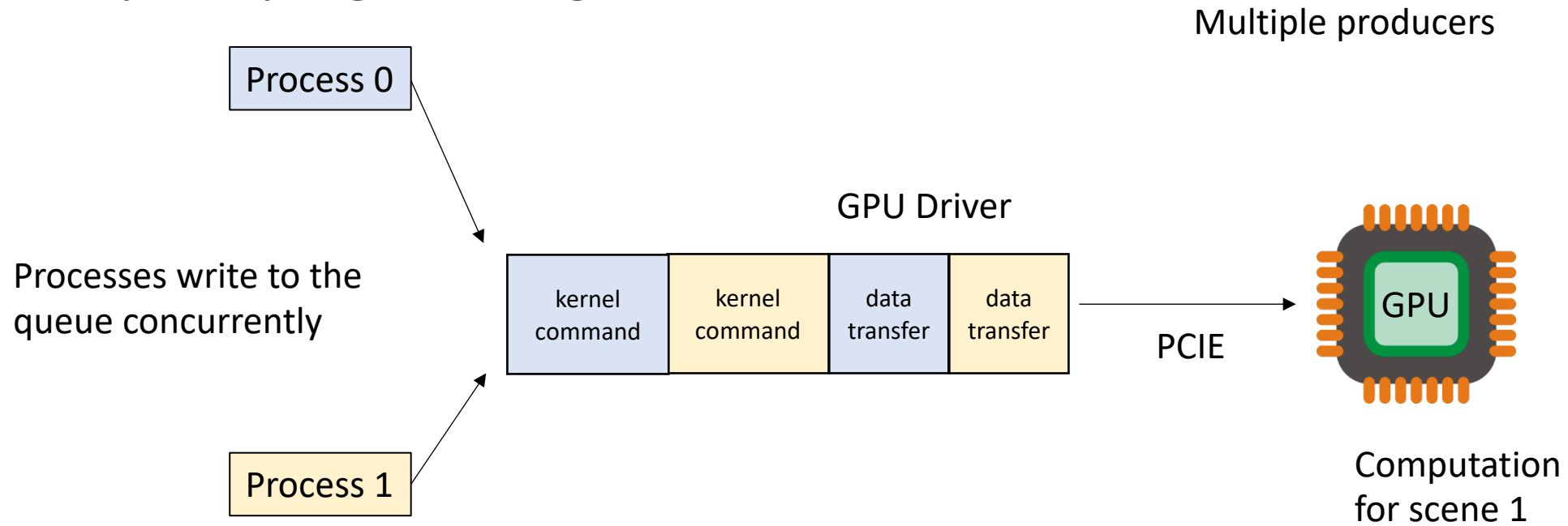
Each process:
loop:
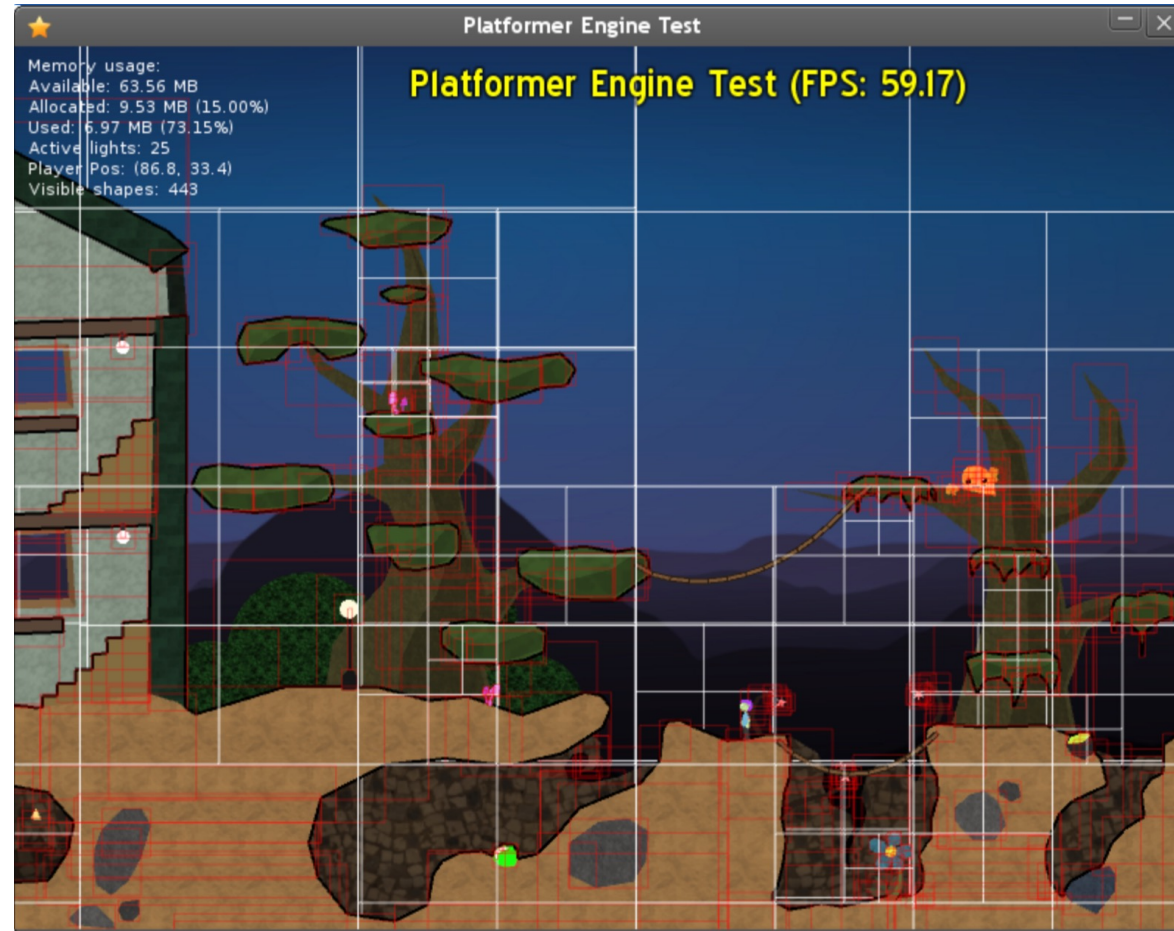   update data (data transfer)
   graphics computation (kernel)

Intro to concurrent objects

- Prior examples have been infrastructural:
  - things happening behind the scenes, drivers, OS, etc.

- They also exist in standalone applications

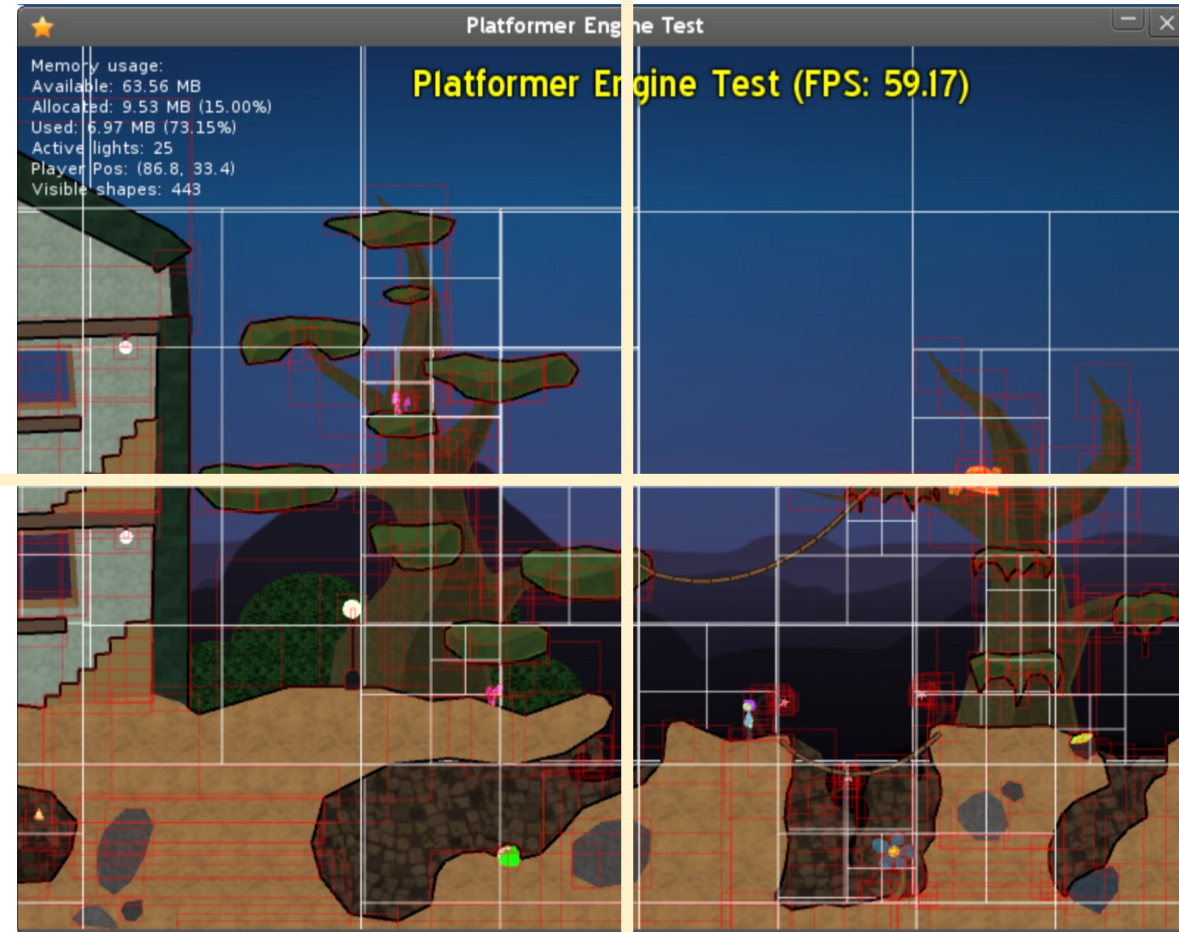# Shared memory concurrent objects

- Quadtree/Octree

Shared memory concurrent objects

- Quadtree/Octree

recursively divide
the scene giving more
detail to "interesting"
areas

Shared memory concurrent objects

- Quadtree/Octree

recursively divide
the scene giving more
detail to "interesting"
areas

Octree example

- From GTC 2012
  - Simulation of 2 galaxies colliding
  - 280K stars

https://www.youtube.com/watch?v=aByz-mxOXJM

Octree example

- From GTC 2012
  - Simulation of 2 galaxies colliding
  - 280K stars

https://www.youtube.com/watch?v=aByz-mxOXJM
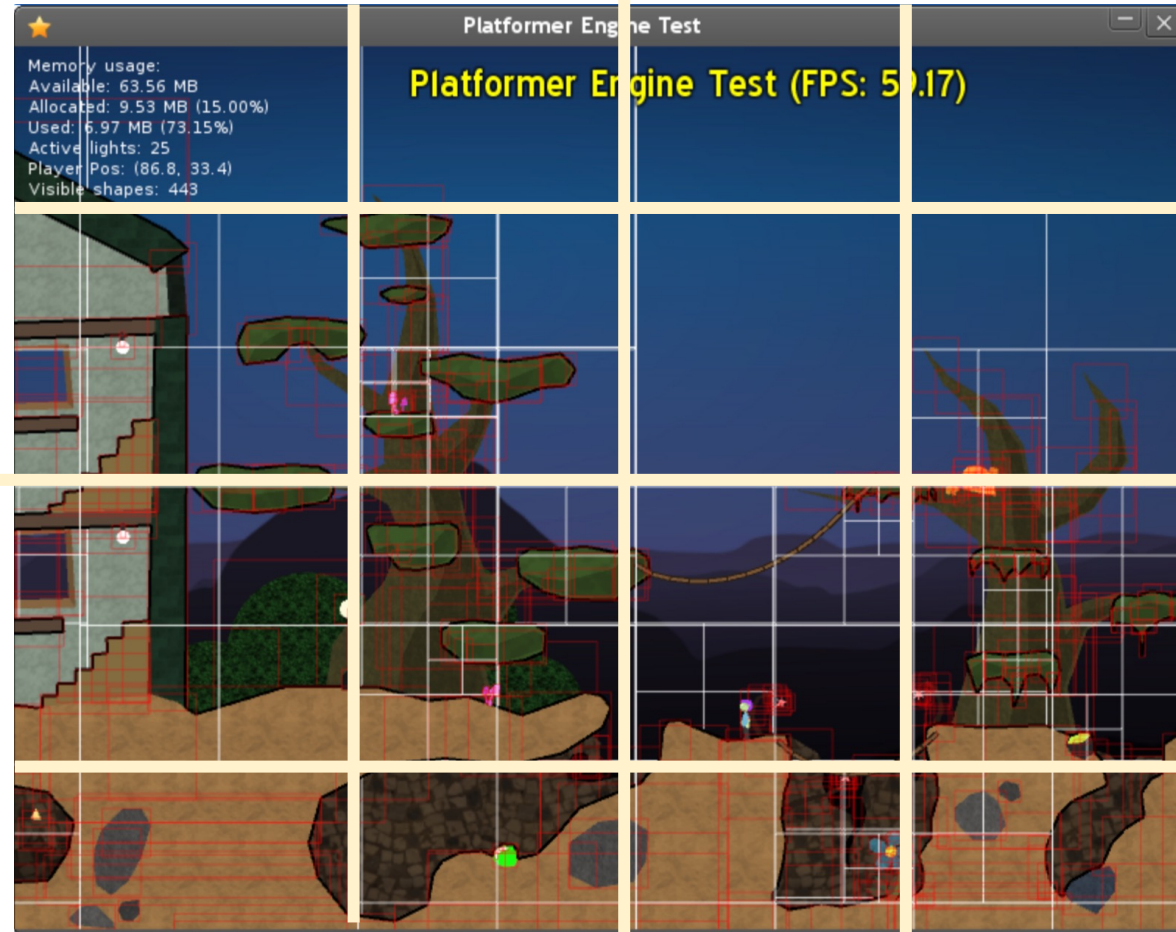
Take away

- Concurrent data structures are everywhere!
  - Infrastructure
  - Applications

Schedule

- Intro to concurrent data structures
- **Bank account example**
- Specification: Sequential consistency

Bank account example

```
int tylers_account = 0;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account -= 1;
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account += 1;
}
```

Bank account example

global variables:

```
int tylers_account = 0;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account -= 1;
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account += 1;
}
```

We might decide to wrap my bank account in an object

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

Bank account example

We might decide to wrap my bank account in an object

global variables:

```
bank_account tylers_account;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account.buy_coffee();
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account.get_paid();
}
```

```cpp
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

Bank account example

We might decide to wrap my bank account in an object

global variables:

```
bank_account tylers_account;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account.buy_coffee();
}
```

what happens if we run these concurrently?

Example

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account.get_paid();
}
```

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

Bank account example

We might decide to wrap my bank account in an object

global variables:

```
bank_account tylers_account;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account.buy_coffee();
}
```

what happens if we run these concurrently?

Example

C++ will not magically make your objects concurrent!

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account.get_paid();
}
```

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

The object is not "thread safe"

Bank account example

We might decide to wrap my bank account in an object

global variables:

```
bank_account tylers_account;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account.buy_coffee();
}
```

First solution:
The client (user of the object) can use locks.

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account.get_paid();
}
```

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

The object is not "thread safe"

global variables:

```
bank_account tylers_account;
mutex m;
```

what if you have
multiple objects?

We might decide to wrap my bank
account in an object

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    m.lock();
    tylers_account.buy_coffee();
    m.unlock();
}
```

First solution:
The client (user
of the object) can
use locks.

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    m.lock();
    tylers_account.get_paid();
    m.unlock();
}
```

```
class bank_account {
  public:
    bank_account() {
        balance = 0;
    }

    void buy_coffee() {
        balance -= 1;
    }

    void get_paid() {
        balance += 1;
    }

  private:
    int balance;
};
```

The object is not "thread safe"

```
bank_account tylers_account;
mutex m;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    m.lock();
    tylers_account.buy_coffee();
    m.unlock();
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    m.lock();
    tylers_account.get_paid();
    m.unlock();
}
```

First solution:
The client (user of the object) can use locks.

client has to manage locks

We might decide to wrap my bank account in an object

```
class bank_account {
  public:
    bank_account() {
        balance = 0;
    }

    void buy_coffee() {
        balance -= 1;
    }

    void get_paid() {
        balance += 1;
    }

  private:
    int balance;
};
```

The object is not "thread safe"

Bank account example

global variables:

```
bank_account tylers_account;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account.buy_coffee();
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account.get_paid();
}
```

we can encapsulate a mutex in the object.

The API stays the same!

```cpp
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```

Thread safe objects

- An object is thread-safe if you can call it concurrently

- Otherwise you must provide your own locks!

Lock free programming

- An object is "lock free" if it is:
  - thread safe
  - does not use a lock in its underlying implementation.

- We can make a lock free bank account

```
atomic_fetch_add(atomic_int * addr, int value) {
    int tmp = *addr; // read
    tmp += value;     // modify
    *addr = tmp;      // write
}
```

Bank account example

global variables:

```
bank_account tylers_account;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account.buy_coffee();
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account.get_paid();
}
```

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```

Bank account example

global variables:

```
bank_account tylers_account;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account.buy_coffee();
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account.get_paid();
}
```

```cpp
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    atomic_int balance;
    mutex m;
};
```

Bank account example

global variables:

```
bank_account tylers_account;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account.buy_coffee();
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account.get_paid();
}
```

```cpp
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {

      balance -= 1;

    }

    void get_paid() {

      balance += 1;

    }

  private:
    atomic_int balance;
};
```

Bank account example

global variables:

```
bank_account tylers_account;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
    tylers_account.buy_coffee();
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
    tylers_account.get_paid();
}
```

```cpp
class bank_account {
  public:
    bank_account() {
        balance = 0;
    }

    void buy_coffee() {
        atomic_fetch_add(&balance, -1);
    }

    void get_paid() {
        atomic_fetch_add(&balance, 1);
    }

  private:
    atomic_int balance;
};
```

How does it perform

How does it perform

- Noticeably better!
  - Mutexes reduce parallelism
  - Mutexes require many RMW operations

- Straight forward to do with the bank account, we will apply this to more objects
  - This performance matters in frameworks!

3 dimensions for concurrent objects

- **Correctness**:
  - How should concurrent objects behave (Specification)

- **Performance**:
  - How to make things fast fast fast!

- **Fairness**:
  - Under what conditions can concurrent objects deadlock

Schedule

- Intro to concurrent data structures
- Bank account example
- **Specification: Sequential consistency**

Lets think about a Queue

What is a queue?

We consider 2 API functions:
- enq(value v) - enqueues the value v
- deq() - returns the value at the front of the queue

```
Queue<int> q;
q.enq(6);
int t = q.deq();
```

```
Queue<int> q;
q.enq(6);
q.enq(7);
int t = q.deq();
```

```
Queue<int> q;
q.enq(6);
q.enq(7);
int t = q.deq();
int t1 = q.deq();
```

Lets think about a Queue

What is a queue?

We consider 2 API functions:
- enq(value v) - enqueues the value v
- deq() - returns the value at the front of the queue

```
Queue<int> q;
int t = q.deq();
```

Let's say: *Error value of 0*

Lets think about a Queue

This is called a sequential specification:

The sequential specification is nice! We want to base our concurrent specification on the sequential specification

We will have to deal with the non-determinism of concurrency

Thinking about a concurrent queue

```
Queue<int> q;
q.enq(6);
q.enq(7);
int t = q.deq();
```

Thinking about a concurrent queue

Global variable:

```
CQueue<int> q;
```

Lets call our concurrent queue "CQueue"

Thread 0:
```
q.enq(6);
q.enq(7);
int t = q.deq();
```

Thinking about a concurrent queue

Global variable:
```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t = q.deq();
```

what can be stored in t after this concurrent program?

Thinking about a concurrent queue

Global variable:
```
CQueue<int> q;
```

```
Thread 0:
q.enq(6);
q.enq(7);
```

```
Thread 1:
int t = q.deq();
```

what can be stored in t after this concurrent program?

Can t be 256?

Thinking about a concurrent queue

Global variable:
```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t = q.deq();
```

what can be stored in t after this concurrent program?

Can t be 256? it should be one of {None, 6, 7}

Global variable:

```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Construct a sequential timeline of API calls
Any sequence is valid:

Thread 1:
```
int t = q.deq();
```
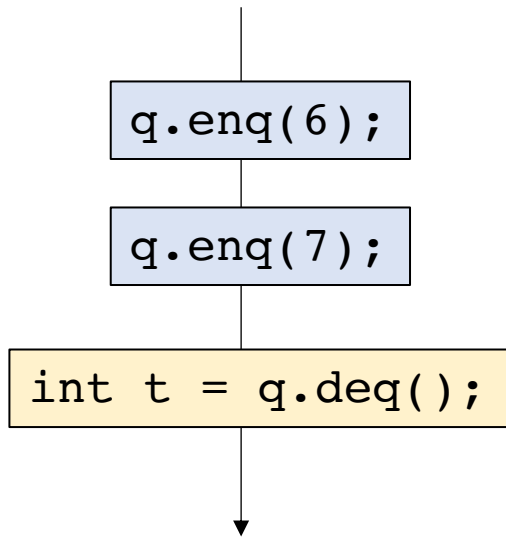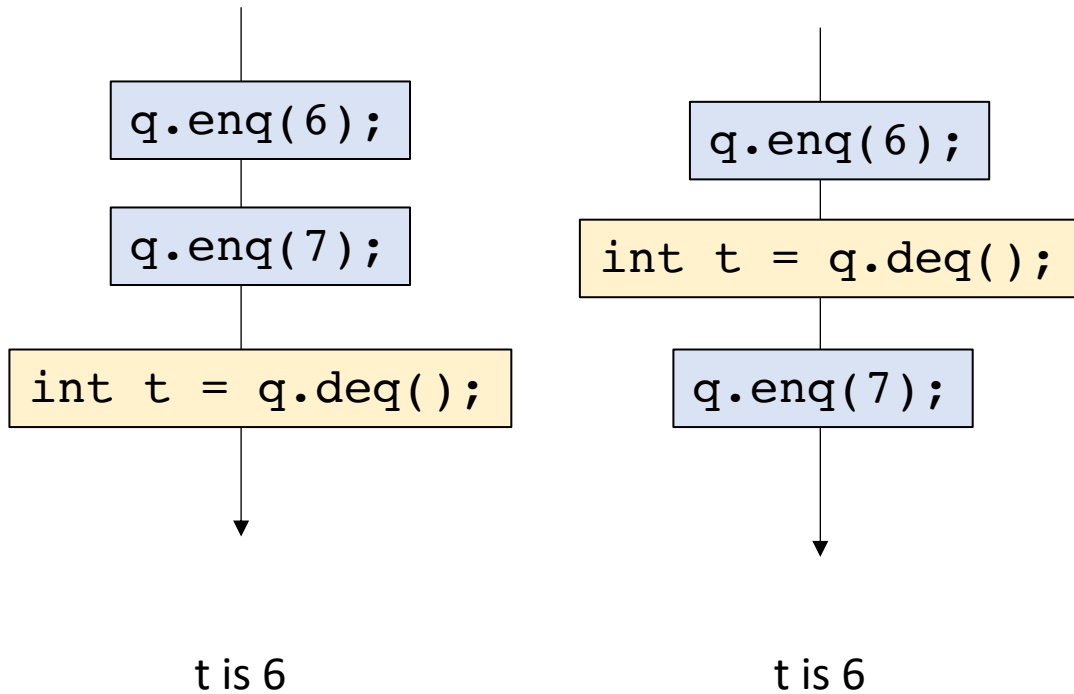
Global variable:

```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t = q.deq();
```

Construct a sequential timeline of API calls
Any sequence is valid:

```
q.enq(6);
```

```
q.enq(7);
```

```
int t = q.deq();
```
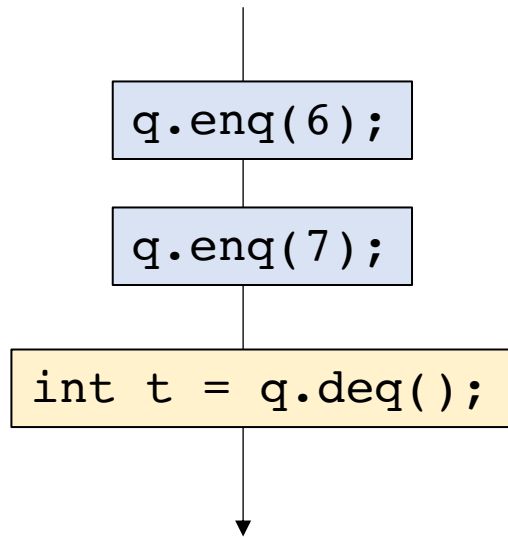
t is 6

Global variable:

```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t = q.deq();
```

Construct a sequential timeline of API calls
Any sequence is valid:

```
q.enq(6);

q.enq(7);

int t = q.deq();
```

t is 6

```
q.enq(6);

int t = q.deq();

q.enq(7);
```

t is 6

Global variable:
```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t = q.deq();
```

Construct a sequential timeline of API calls
Any sequence is valid:

```
q.enq(6);

q.enq(7);

int t = q.deq();
```
t is 6

```
q.enq(6);

int t = q.deq();

q.enq(7);
```
t is 6
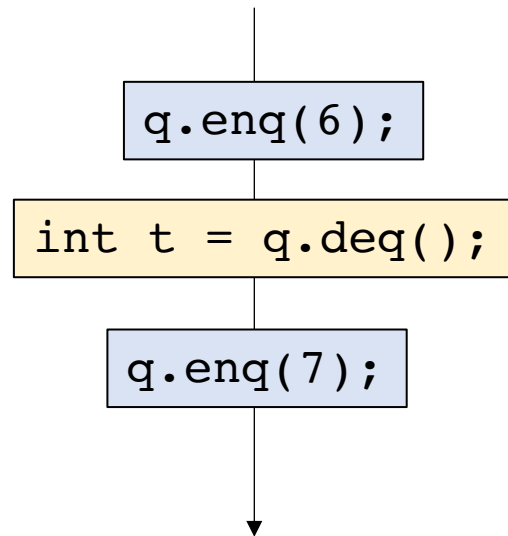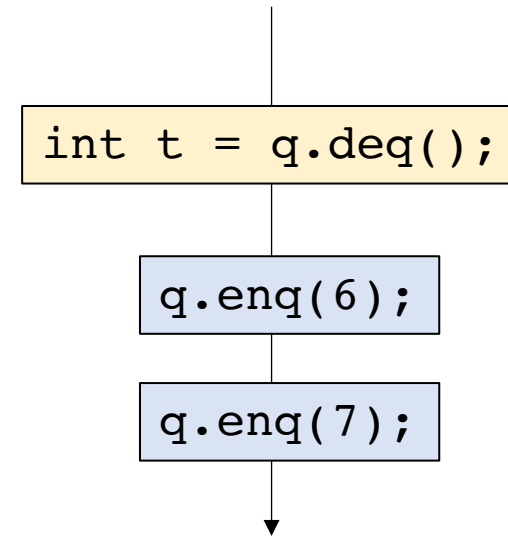
```
int t = q.deq();

q.enq(6);

q.enq(7);
```
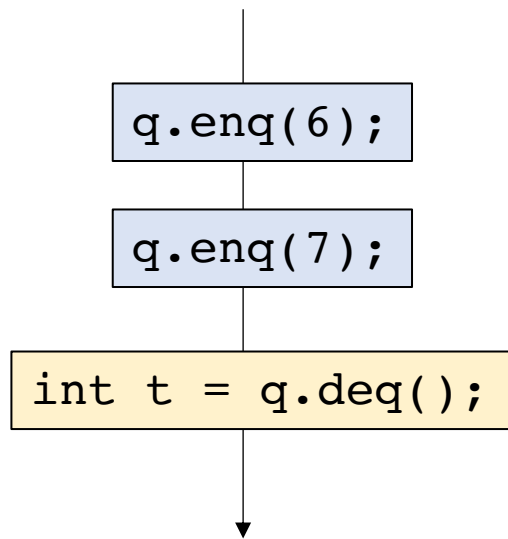t is None

Global variable:
```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t = q.deq();
```

Construct a sequential timeline of API calls
Any sequence is valid:


```
q.enq(6);
```
```
q.enq(7);
```
```
int t = q.deq();
```
t is 6

```
q.enq(6);
```
```
int t = q.deq();
```
```
q.enq(7);
```
t is 6

```
int t = q.deq();
```
```
q.enq(6);
```
```
q.enq(7);
```
t is None

Can t ever be 7?

Global variable:
`CQueue<int> q;`

Thread 0:
`q.enq(6);`
`q.enq(7);`

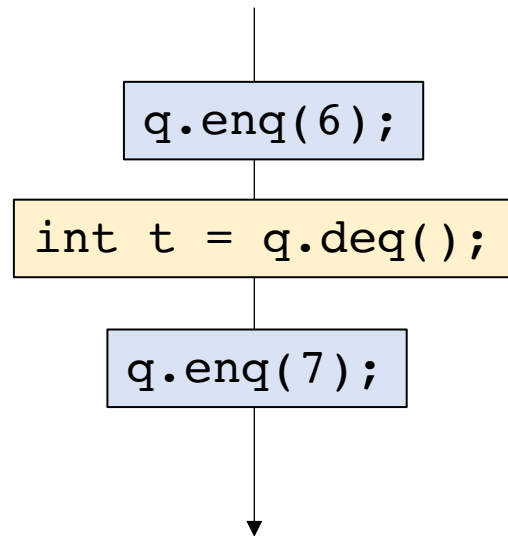*Thread 1:*
`int t = q.deq();`
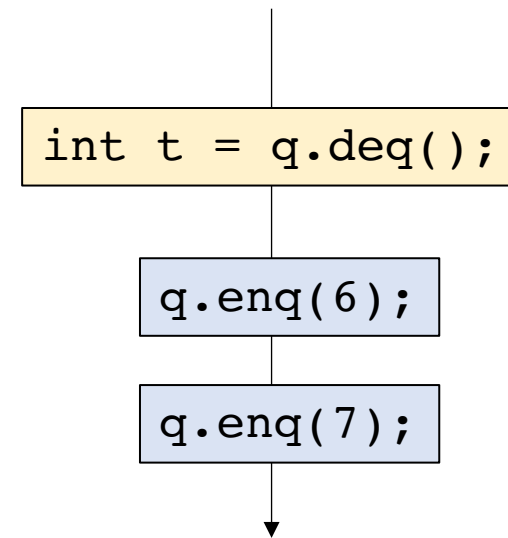
Construct a sequential timeline of API calls
Any sequence is valid:

Can t ever be 7?

Global variable:
```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t = q.deq();
```

Construct a sequential timeline of API calls
Any sequence is valid:

```
q.enq(7);
```

Can t ever be 7?

Global variable:

`CQueue<int> q;`

Thread 0:
`q.enq(6);`
`q.enq(7);`

Thread 1:
`int t = q.deq();`

Construct a sequential timeline of API calls
Any sequence is valid:

`q.enq(7);`

`int t = q.deq();`

Can t ever be 7?

Global variable:

```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t = q.deq();
```

Construct a sequential timeline of API calls
Any sequence is valid:

```
q.enq(7);
```

```
int t = q.deq();
```

```
q.enq(6);
```

Can t ever be 7?

Global variable:
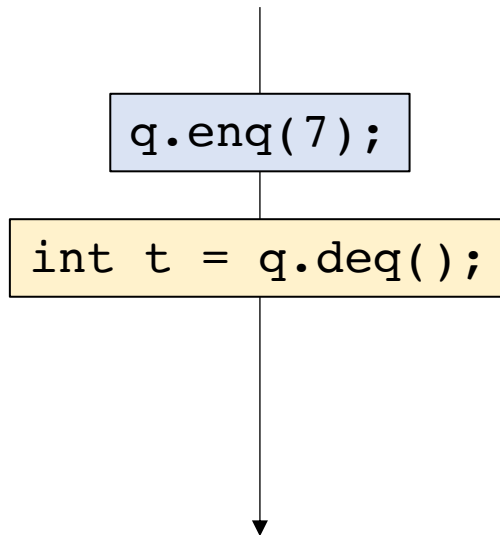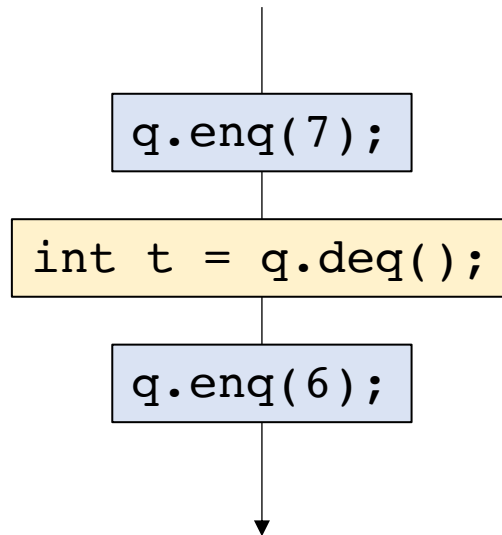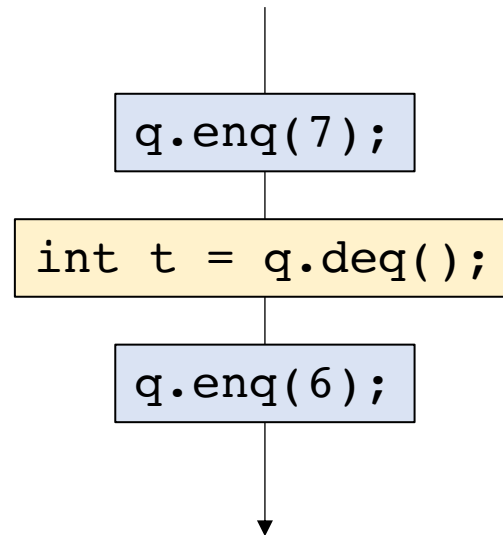
```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t = q.deq();
```

Construct a sequential timeline of API calls
Any sequence is valid:

*The events of Thread 0
don't appear in the same
order of the program!*

This should not be allowed!

```
q.enq(7);
```

```
int t = q.deq();
```

```
q.enq(6);
```

*Can t ever
be 7?*

Sequential Consistency

- Valid executions correspond a sequentialization of object method calls

- The sequentialization must respect per-thread "program order", the order in which the object method calls occur in the thread

- Events across threads can interleave in any way possible

Sequential Consistency

- Valid executions correspond a sequentialization of object method calls

- The sequentialization must respect per-thread "program order", the order in which the object method calls occur in the thread

- Events across threads can interleave in any way possible

How many possible interleavings?
Combinatorics question:

if Thread 0 has N events
if Thread 1 has M events

$$\frac{(N + M)!}{N! M!}$$

Sequential Consistency

How many possible interleavings?
Combinatorics question:

if Thread 0 has N events
if Thread 1 has M events

$$\frac{(N + M)!}{N!\,M!}$$

Reminder that N and M are events, not instructions

Sequential Consistency

```
for (int j = 0; j < HOURS; j++) {
    tylers_account += 1;
}
```

How many possible interleavings?
Combinatorics question:

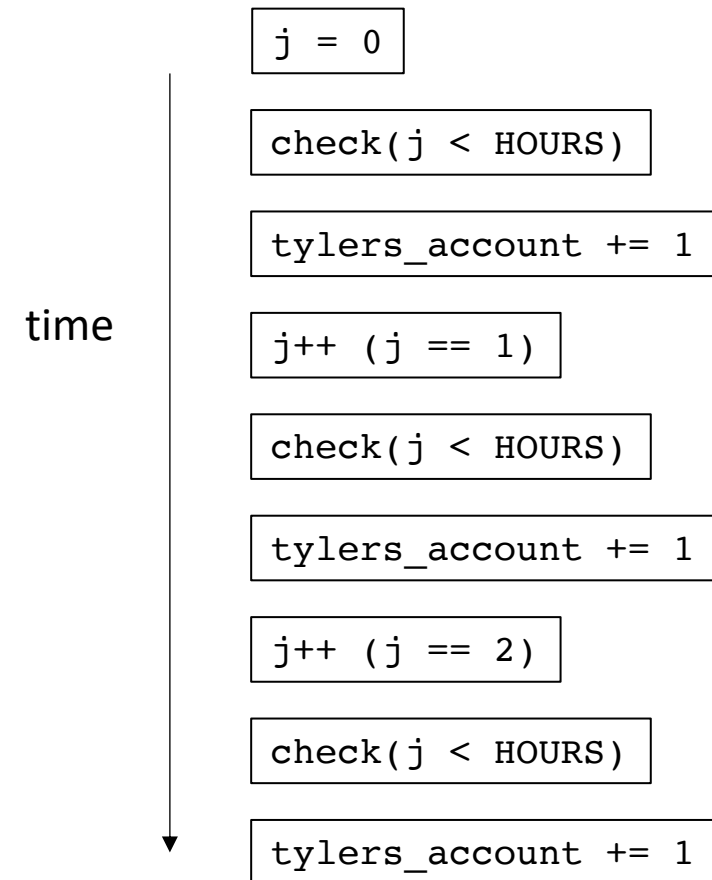if Thread 0 has N events
if Thread 1 has M events

$$\frac{(N + M)!}{N!\,M!}$$

Reminder that N and M are events, not instructions

If N and M execute 150 events each, there are more
possible executions than particles in the observable universe!

j = 0

check(j < HOURS)

tylers_account += 1

time

j++ (j == 1)

check(j < HOURS)

tylers_account += 1

j++ (j == 2)

check(j < HOURS)

tylers_account += 1

Don't think about all possible interleavings!

- Higher-level reasoning:
  - I get paid 100 times and buy 100 coffees, I should break even
  - If you enqueue 100 elements to a queue, you should be able to dequeue 100 elements

- Reason about a specific outcome
  - Find an interleaving that allows the outcome
  - Find a counter example

Reasoning about concurrent objects

To show that an outcome is possible, simply construct the sequential sequence

Global variable:

```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t0 = q.deq();
int t1 = q.deq();
```

Can t0 == 0 and t1 == 6?

Global variable:

```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t0 = q.deq();
int t1 = q.deq();
```

```
q.enq(6);
```

```
q.enq(7);
```

```
int t0 = q.deq();
```

```
int t1 = q.deq();
```

Can t0 == 0 and t1 == 6?

Global variable:
```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t0 = q.deq();
int t1 = q.deq();
```

```
int t0 = q.deq();
```

```
q.enq(6);
```

```
q.enq(7);
```

```
int t1 = q.deq();
```

Can t0 == 0 and t1 == 6?

Valid execution!

Are there others?

Global variable:

```
CQueue<int> q;
```

Lets do another!

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t0 = q.deq();
int t1 = q.deq();
```

Can t0 == 6 and t1 == 7?

Global variable:

```
CQueue<int> q;
```

Lets do another!

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t0 = q.deq();
int t1 = q.deq();
```

```
q.enq(6);
```

```
q.enq(7);
```

```
int t0 = q.deq();
```

```
int t1 = q.deq();
```

Can t0 == 6 and t1 == 7?

Global variable:

```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t0 = q.deq();
int t1 = q.deq();
```

```
q.enq(6);
```
```
int t0 = q.deq();
```
```
q.enq(7);
```
```
int t1 = q.deq();
```

Found one! Are there others?

Can t0 == 6 and t1 == 7?

Reasoning about concurrent objects

To show that an outcome is possible, simply construct the sequential sequence

To show that an outcome is **impossible** show that there is no possible sequential sequence
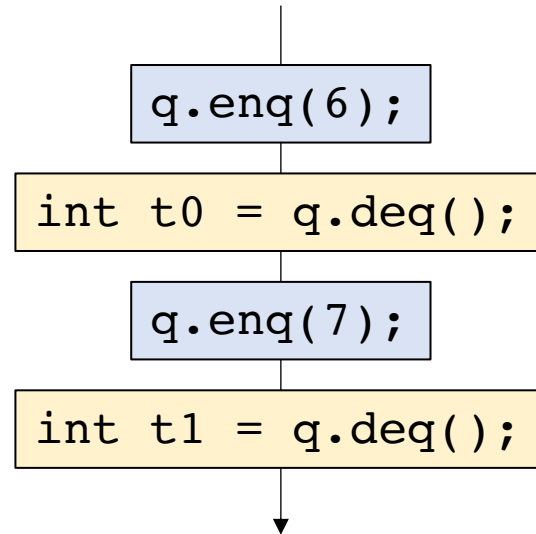
Global variable:

```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t0 = q.deq();
int t1 = q.deq();
```

Can t0 == 0 and t1 == 7?

Global variable:

```
CQueue<int> q;
```

Thread 0:
```
q.enq(6);
q.enq(7);
```

Thread 1:
```
int t0 = q.deq();
int t1 = q.deq();
```

```
q.enq(6);
```

```
q.enq(7);
```

```
int t0 = q.deq();
```

```
int t1 = q.deq();
```

Can t0 == 0 and t1 == 7?

Global variable:
```
CQueue<int> q;
```

```
int t0 = q.deq();
```

```
q.enq(7);
```

```
int t1 = q.deq();
```

```
q.enq(6);
```

No place for this event to go!

Can t0 == 0 and t1 == 7?

One more example

Global variable:

```
CStack<int> s;
```

Thread 0:
```
s.enq(7);
int t0 = q.dec();
```

Thread 1:
```
int t1 = q.dec();
```

Is it possible for both t0 and t1 to be 0 at the end?

Global variable:

```
CQueue<int> s;
```

Thread 0:
```
s.enq(7);
int t0 = q.dec();
```

Thread 1:
```
int t1 = q.dec();
```

```
int t1 = q.dec();
```

```
q.enq(7);
```

```
int t0 = q.deq();
```

Is it possible for both t0 and t1 to be 0 at the end?

Do we have our specification?

- Is sequential consistency a good enough specification for concurrent objects?

- It's a good first step, but relative timing interacts strangely with absolute time.

- We will need something stronger.

Next time

- See you on Friday!

- Work on HW 2
  - Visit office hours or ask questions on Piazza if you need to!